CTC++ Bitcov add-on

User's Guide Version v0.33

Verifysoft TECHNOLOGY

Softwaretest- und Analysetools

CTC++ Bitcov add-on User's Guide, v0.33

(Version of July 29, 2010)

Previous document versions were in text format, divided up into *readme.txt*, *qstart.txt* and *version.txt* files, and also further documentation about each component of our software, contained in text and html files.

Copyright © 2006-2010 Verifysoft Technology GmbH
Copyright © 2006-2010 Testwell Oy
All distinctive marks are properties of their respective holders.

For further information, please contact **Verifysoft Technology GmbH**:

- mail: info@verifysoft.com
- phone (Germany): +49(0)781 63 92 027
- phone (France): +33(0)3 68 33 58 84

Author: Christophe Sourisse (Verifysoft Technology GmbH)

Testwell Oy

Hermiankatu 8
FI-33720 Tampere
Finland

Verifysoft Technology GmbH

Technologiepark, In der Spöck 10
D-77656 Offenburg
Germany

Contents

1 Introducing CTC++Bitcov add-on **4**
- 1.1 About CTC++Bitcov add-on . . . 4
- 1.2 About Test Coverage in Testwell CTC++with CTC++Bitcov add-on . . . 4
- 1.3 About this version of CTC++Bitcov add-on . . . 5
- 1.4 About previous versions of CTC++Bitcov add-on . . . 5

2 Installing CTC++Bitcov add-on **8**
- 2.1 Composition of CTC++Bitcov add-on package . . . 8
- 2.2 Basic installation . . . 9
- 2.3 Common adjustments . . . 9

3 Tutorial Example **11**
- 3.1 Introduction . . . 11
- 3.2 Instrumentation process . . . 11
- 3.3 Getting back the execution counters . . . 12
  - 3.3.1 Manual way . . . 12
  - 3.3.2 Using `any2mem` CTC++Bitcov utility . . . 13
  - 3.3.3 Conclusion . . . 16
- 3.4 About `MON.aux` file and execution counters storage in `CTC_array[]` . . . 18
- 3.5 Recapitulation of CTC++Bitcov execution process . . . 19

4 Walkthrough with `prime` example **24**
- 4.1 Manual way (using a debugger) . . . 24
- 4.2 Automatic transfer (using `any2mem` utility) . . . 25

5 How to work with real (32-bits) counters **28**

A CTC++Bitcov utilities man pages **29**
- A.1 Man page: `any2mem` utility . . . 29
- A.2 Man page: `ctc2static` utility . . . 29
- A.3 Man page: `dmp2txt` utility . . . 29
- A.4 Man page: `montxt2dmp` utility . . . 31

B Table of `-DCTC_*` macro definitions **32**

List of Figures **33**

Glossary **35**

Index **37**

1 Introducing CTC++ Bitcov add-on

1.1 About CTC++ Bitcov add-on

Nowadays, as programs developed are bigger and bigger, and more and more complex, the risk they contain some errors doesn't keep growing. That's why Software Testing becomes a priority for many developing societies.

In 1989, the companies Nokia Data Systems Oy / ICL Personal Systems Oy published a first solution: a powerful instrumentation-based tool providing test coverage and dynamic analysis for programs written in C, called CTC. It has then been extended to programs written in C++ in 1991, so that its name changed to CTC++, and at the end of 1993, Testwell took over the development of CTC++.

Since 1994, Testwell has released several enhancements of this tool, especially a CTC++ Host-Target add-on enabling developers to work on embedded targets. By storing execution counters during testes, Testwell CTC++ calculates, for each function, source file and the whole program, a TER, showing how many percent of source code have already been tested. The Testwell CTC++ Test Coverage Analyzer for C/C++ is, with its classical Host-Target package, a very good solution in most cases, even when program testing should be done on embedded targets, which often have only a few kilo of available RAM.

However, some target systems have an insufficient RAM, or not any filesystem, which makes that Testwell CTC++ cannot run its testing softwares on it, or just can run it for the lowest coverage levels and so test coverage remains insufficient. This is where CTC++ Bitcov add-on goes into action: with its help, we should be able to use Testwell CTC++ with less then 20 times of RAM compare to the classical Host-Target arrangement, what means just a few or a few dozens bytes RAM.

Thanks to its architecture neutral approach, CTC++ Bitcov add-on makes it possible to absolutely minimise the memory consumption during Testwell CTC++ execution, and provides its using on smallest embedded targets and Microcontrollers.

1.2 About Test Coverage in Testwell CTC++ with CTC++ Bitcov add-on

By using CTC++ Bitcov Add-on, all test coverage levels are left provided. As a reminder, these are mainly:

- **Function coverage**: examins which functions in the source code have actually been called, how many times they have been called and how many times their closing brace has been reached;

- **Decision coverage**: examins for each function whether the decisions in the code have actually been evaluated both to *true* and *false*;

- **Condition coverage**: examins in each function, for each decision, whether the atomic conditions of this decision have actually been evaluated both to *true* and *false*;
- **Modified condition/decision coverage**: examins in each function, for each decision, whether the atomic conditions of this decision have actually been evaluated both to *true* and *false* affecting independently the decision's outcome;
- **Multicondition coverage**: examins in each function, for each decision, whether each possible combination for its evaluation has actaully been evaluated both to *true* or *false*.

Do not forget that these are stronger and stronger test coverage levels: function coverage is included in decision coverage, which is itself included in multicondition coverage. For deeper information about test coverage levels, have a look at the CTC++ User's Guide, section "About Test Coverage in CTC++".

1.3 About this version of CTC++ Bitcov add-on

This CTC++ Bitcov add-on Users Guide describes the version level v0.33 of the tool (July 2010). The previous version of this documentation was written in June 2010.

This User's Guide mainly contains a tutorial example of CTC++ Bitcov add-on usage and some of the reports that it generates. This example is the same as the one employed in last version (v6.5.6) of Testwell CTC++ User's Guide.

With this new v0.33 version of CTC++ Bitcov add-on, the `CTC_SAFE` macro, which prevent users from execution counters' overflow (for 8-bit counters), was re-added and can now be used with this add-on.

This v0.33 version also contains some other enhancements, which are needed in some (coming) special use cases. They may not have much effect to a normal user. Also some bugs were corrected as usual. The `version.txt` file in the delivery package describes all the changes in more detail.

1.4 About previous versions of CTC++ Bitcov add-on

The first version of CTC++ Bitcov add-on was released in January 2006 (v0.1, originaly un-numbered).

In the v0.2 version of the tool, delivered in June 2006, were added a textual file `version.txt` describing versions' history and `any2mem` helper scripts. Moreover, `hostmain` utility was rebuild to run with Testwell CTC++ 6.3.x versions and `ctc2static` tool use became more secured.

In July 2006, the main new enhancements for v0.8.x were the introduction of `diagnosis.txt`

and `qstart.txt` text files respectively to solve common problems and to give a quick way of using CTC++ Bitcov add-on utilities. Besides, some support/workarounds for Intel's ic86 compilers were added.

In August 2006, the new v0.9 was fitted to Metrowerks by modifying `ctc2static` utility. The stack size problem with `dmp2mon` utility (with more than 1000 files projects using recursivity) was also solved.

With v0.9.1 version released in September 2006, CTC++ include files were rearranged and documentation was updated.

With v0.9.4 version released in November 2006, script files were rearranged for maintenance reasons.

In v0.9.6 version released in December 2006 was added quick check for consitency of the dumped file, to ensure that each function which end is reached has previously been started...

In March 2007, CTC++ Bitcov add-on v0.9.7 (especially `hostmain` utility) was updated to CTC++ v.6.3 level and `dmp2mon` was recompiled on Linux 32-bit platform (was 64-bit in 0.9.6). In addition, applications now can be re-compiled, if the amount of counters has not changed, without deleting `MON.aux` file. Else, user positively fails to compile.

In January 2008, CTC++ Bitcov add-on v0.9.8 was updated to CTC++ v.6.4 level and to CTC++ Host-Target package v3.2 level.

In February 2008, CTC++ Bitcov add-on v0.9.8 was updated to CTC++ v.6.5 level, and `--version` and `--usage` options were added to the tools.

With v0.9.9.3 version released in August 2008, user disposes from now on of an installer for Windows, and of `hex2bin` and `mot2bin` new CTC++ Bitcov utilities.

With v0.10 version released in October 2008, `hex2bin` and `mot2bin` utilities were removed and `dmp2mon` executable was replaced with `dmp2txt` script, which can now read directly Intel, Motorola and also Tektronix dump files. Documentation was also updated.

In v0.18 and v0.19 versions respectively released in November 2008 and January 2009, some specific Windows adjustments were done.

Also in January 2009, the v.20 version prevents running as regular user on Vista/Windows if Testwell CTC++ is installed in an UAC directory.

In v0.21 version released in March 2009, `hostmain` was removed for compatibility purposes (there was an error while piping some results to it) and can be replaced with `ctc2dat` bundled with CTC++ v6.5.4 or higher.

In v0.22 and v0.23 versions released in June 2009, some files and folders were reorganized, documentation was updated and header files were also reorganized.

With v0.26 version released in July 2009, CTC++ Bitcov add-on was fitted to Wine.

With v0.27 version released in July 2009, `ctc2static` utility no longer creates entries in `MON.aux` file, if number of test-points are zero.

With v0.28 version released in Febrary 2010, `any2mem` utility also emits a `ctc_init()` function, which can set `CTC_array[]` to zero.

With v0.30 version released in March 2010, `any2mem` utility outputs C code proper prototyped.

Eventually, in June 2010, `montxt2dmp` utility was added for the host side to the v0.31 version, and extended (with `dmp2txt` utility) to native endian in the v0.32 version.

2 Installing CTC++ Bitcov add-on

2.1 Composition of CTC++ Bitcov add-on package

A standard CTC++ Bitcov add-on package will be composed of:

- `any2mem`
  Script of `any2mem` utility, calculating needed `CTC_array[]`, especially its size. More details at appendix A.1 (on page 29).

- `ctc2static`
  Script of `ctc2static` utility (`RUN_AFTER_INSTR` script). More details at appendix A.2 (on page 29).

- `dmp2txt`
  Script of `dmp2txt` utility, converting memory dump to text, as expected by `ctc2dat`. More details at appendix A.3 (on page 29).

- `montxt2dmp`
  Script of `montxt2dmp` utility, converting `MON.txt` file to memory dump file, as expected by `dmp2txt`. More details at appendix A.4 (on page 31).

- `doc/`
  Here you can find some text and HTML documentation, including the man pages content (about CTC++ Bitcov utilities and files generated by using CTC++ Bitcov add-on), and some additionnal information about using CTC++ Bitcov add-on on "Huge Targets".

- `ctcbits.h`
  This is a header file, complement of `ctc.h` for CTC++ Bitcov add-on, which redefines some of the dynamic (runtime) macros away for the static (compile time) approach.

- `man/`
  There are some man pages, concerning CTC++ Bitcov utilities (`any2mem`, `ctc2static`, `dmp2txt` and `montxt2dmp`) and files generated by using CTC++ Bitcov add-on (`MON.aux`, `MON.dmp` and `MON.hex`).

- `install` and `install.bat`
  The installation files (for Windows users).

- `makefile`
  The Makefile (for Unix/Linux users).

- `qstart.txt`
  Just a quick walk through with default settings/places to start using CTC++ Bitcov add-on.

- `readme.txt`
  A deeper detailed file than `qstart.txt`, containing further information about the execution of Testwell CTC++ with our CTC++ Bitcov add-on.

- `Setup.exe`
  An alternative to `install` files if you need Administrator privilegs (Windows Vista or Windows 7).

- `version.txt`
  This file contains version change history of CTC++.

2.2 Basic installation

The detailed machine specific installation procedure is described in separate documentation. You can especially refer to `qstart.txt` for a quick installation, and `readme.txt` for deeper detailed explanations.

On Windows platform run:

```
Setup.exe
```

in the directory of CTC++ Bitcov package.

On Unix/Linux platform run:

```
make install
```

in a terminal, in the same way as you have installed Testwell CTC++ itself.

Depending on the installation directory, administration rights may be needed for running the installation script.

See also the 2.3 section (page 9, "Common adjustments") for more informations about the common settings which should be done before experimenting CTC++ Bitcov add-on.

2.3 Common adjustments

Mainly, you should have to teach to CTC++ it has to call the script `ctc2static` just after having done the instrumentation of the C-preprocessed file and written the instrumented temporary file of the source. This will produce a `MON.aux` file, which content is needed to use the `dmp2txt` CTC++ Bitcov utility, as we will see it at 3.5 section (page 19, "Recapitulation of CTC++ Bitcov execution process").

For deeper information about `ctc2static` and `dmp2txt` CTC++ Bitcov utilities, refer respectively to appendix A.2 at page 29 and to appendix A.3 at page 29. The technical background is also deeper explained in CTC++ User's Guide.

On Windows: copy `ctc2static.bat` to `%CTCHOME%`.

In a Unix-like environment, you may just copy `ctc2static` to the `bin` directory of the CTC++ installation (default is `/usr/local/bin/`).

Then, to make this active, add `ctc2static` to the compiler section in your `ctc.ini` file:

```
RUN_AFTER_INSTR = ctc2static
```

In doubt case, add this statement in every (comma separated list) section of a compiler which may be used. On top of that, depending on the installation directory, administration rights may be needed for editing the `ctc.ini` file.

3 Tutorial Example

3.1 Introduction

This section provides for you a typical, concrete and complete example of what a CTC++ Bitcov add-on execution looks like.

Assume we want to test the little `prime` program on a microcontroller-like Target system. The global structure of `prime` is deeper described in the CTC++ User's Guide at section "Tutorial Example".

3.2 Instrumentation process

During an ordinary execution of CTC++ (i.e. in the solely host-based mode of CTC++), we just had to compile our source files with CTC++ Preprocessor, run as many testes as we wanted, during which the execution counters were automatically saved in `MON.dat` binary file, and finally use the CTC++ Postprocessor to produce the Execution Profile Listing:

```
ctcwrap -i m make
-- run testes... --
ctcpost MON.sym MON.dat -p profile.txt
ctc2html -i profile.txt
```

Unfortunately, this process needs memory, sometimes too much to store the data binary file. That is why, in the context of CTC++ Bitcov add-on, we have to do otherwise.

In concrete terms, the main idea of CTC++ Bitcov add-on is to store on the target system (on which we want to test our application) only the essential and necessary data, which are:

- the executable file(s) of our program;
- the execution counters.

Concerning the executable file(s), there is no problem since we just have to do the instrumented compilation on a Host System on which the cross-compiler used for the target runs (for more details, refer to CTC++ User's Guide, at section "Host-Target Testing").

Concerning the file(s) storing the execution counters, if we use Testwell CTC++ utilities (with the *CTC++ Host-Target add-on*) in a traditional way, we may encounter a memory size problem. As described in User's Guide, such a classical use of Testwell CTC++ produces some files containing these execution counters, and all data needed to produce and use it. But this or these file unfortunately require(s) a file system and some RAM available, what many little systems like microcontrollers do not dispose of...

This is mainly this last problem that CTC++ Bitcov add-on tries to solve, which solution is explained hereafter in 3.3 section (page 12, "Getting back the exectution counters").

3.3 Getting back the execution counters

Now assume that the installation process has been successfully done, headers have been added to the good directories and `ctc2static` option has been added for each compilator section in `ctc.ini` file. If that is not the case, refer to 2.3 section (page 9, "Common adjustments").

3.3.1 Manual way

Before compiling and instrumenting the source files, we have to prepare the execution counters saving at the Target system. To this end, we need to add on this Target an array named `CTC_array[]`, initialised with `{0;...;0}`, to hold the counter bits. This should be done declaring as follows:

```
unsigned char CTC_array[0xfff] = {0};
```

at global scope of one of the source files (for example the one containing the `main()` function).

This `CTC_array[]` should be sufficiently big to contain the counters. In order to estimate the needed size of the array, you might just make a "test compile" of your project and run on the generated `MON.sym` file, which contains the files' description, following:

```
ctcpost -L MON.sym | any2mem
```

As it is deeper explained in the `readme.txt` file, round up this estimation and add a little for safety and possible future code extensions.

In our `prime` exemple, we will see after typing this last line in a terminal:

```
Insert following line somewhere in the global scope of your application:
  unsigned char CTC_array[4] = {0};

If you want a compilable output file, redirect any2mem output to a file,
e.g. any2mem > CTC_array.c
```

and so choose about 6 or 7 for our `CTC_array[]` size.

Then, you should somehow transfer the Target array to the Host system, or at the very least, the data it contains. There are several ways to do it, but the most common and flexible solution would be to use a Debug Interface (like *gdb* or *ddd*).

After instrumenting your source files with the `-g` compilation option (which should be

added in our `prime` example in the `makefile`), run the Debug Interface of your choice and set a breakpoint just before the end of your program (in `prime` example, just before the `return 0` in the `main()` function).

Then, run testes through the debugger and, at breakpoint's moment, write down the final content of the `CTC_array[]` and save it to a binary `MON.dmp` file (which will in fact dump one-to-one its successive binary values) by typing in the *gdb* console:

```
dump binary value MON.dmp CTC_array
```

To finish, the following 1 flowchart sums up this manual execution counters saving process.

3.3.2 Using `any2mem` CTC++ Bitcov utility

However, there is another way of doing, without modifying or adding new functions to your source files. To make it possible, we have to use `any2mem` CTC++ Bitcov utility to produce a `CTC_array.c` additionnal source file. This automatically generated file will contain all functions required to save execution counters and produce a `MON.txt` which we will be able to convert easily into a binary `MON.dmp` file, as you will see. For deeper information about `any2mem` CTC++ Bitcov utility, refer to appendix A.1 at page 29.

So first, we have to produce the C file untitled `CTC_array.c`. To do this, we just have to use the `any2mem` CTC++ Bitcov utility as follows:

```
ctcpost -L MON.sym | any2mem -DCTC_RUNTIME > CTC_array.c
```

This is maybe what you could suspect at previous 3.3.1 section (page 12, "Manual way") when you already had to use `ctcpost` utility to get an approximation of `CTC_array[]` size and that terminal said to you *"If you want a compilable output file, redirect any2mem output to a file, e.g. any2mem > CTC_array.c"*.

In this C file, you will find several subroutines and environment variables definitions, which role is to produce at test end a textual `MON.txt` file summing up the final values of execution counters.

Besides, the `-DCTC_RUNTIME` option passed to `any2mem` defines the `CTC_RUNTIME` environment variable in `CTC_array.c` script, and so alows you to use the functions provided by it. If you want to use real counters ("Huge Targets" approach), you also should define CTC_NO_BITS environment variable in this `CTC_array.c` file by adding -DCTC_NO_BITS option when you call `any2mem` utility, as explained in 5 section ("How to work with real (32-bits) counters", page 28).

Do not forget to compile first your application with CTC++ Preprocessor, in order to dispose of a `MON.sym` file which contains a description of your source files needed to use `ctcpost` utility. In `prime` example, the command to type should look like:

```
ctc -i m gcc -c calc.c io.c prime.c
```

with the instrumentation mode of your choice specified by `-i` option.

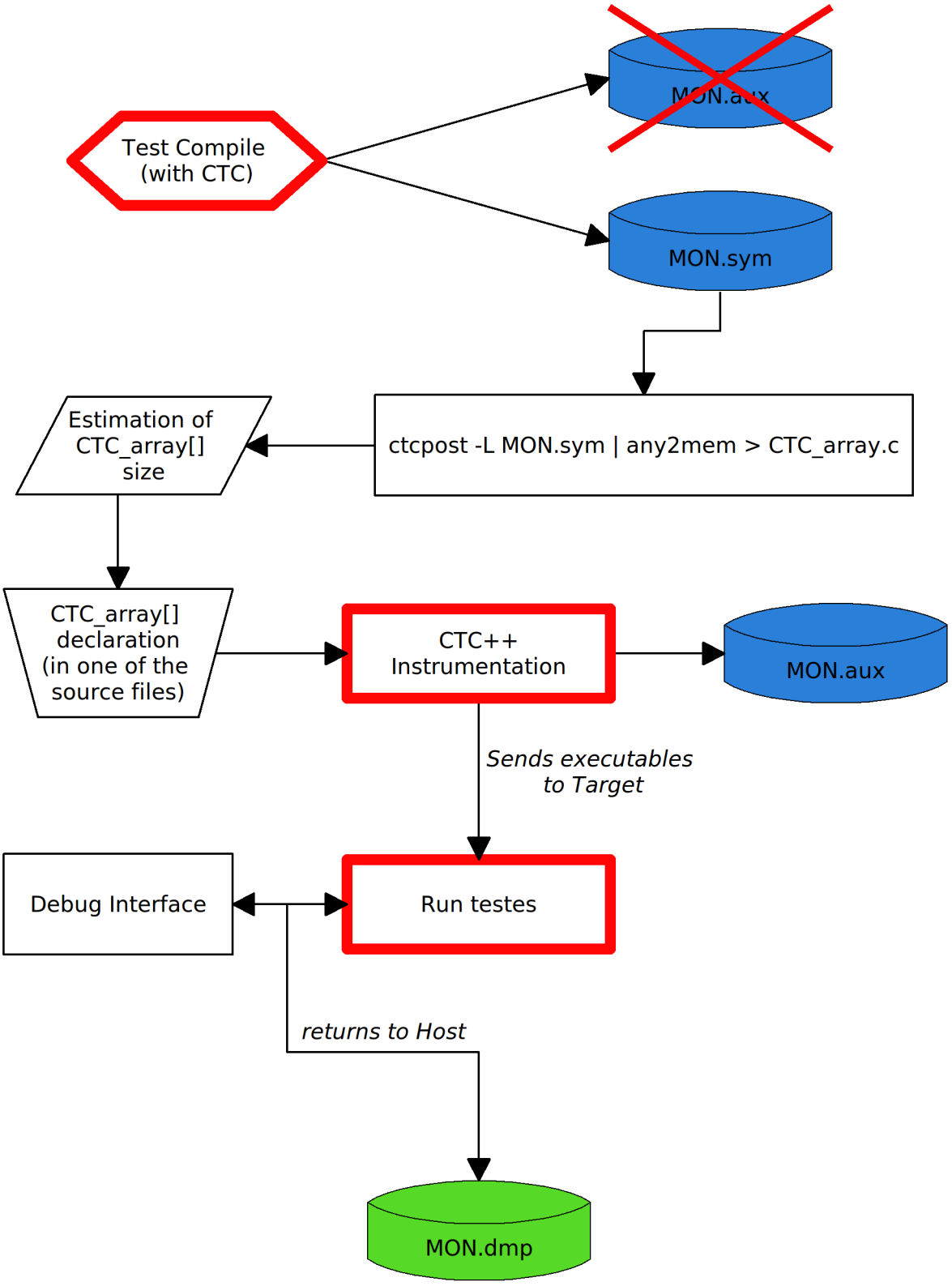

Figure 1: Manual execution counters saving.

Then, compile the CTC_array.c file so produced with your cross-compiler, e.g.:

```
gcc -c CTC_array.c
```

Once that has been done, you have to add some options to your ctc.ini file (which you can find by default at /usr/local/lib/ctc/ direction) at the compiler's section:

- add the executable CTC_array.o file to LIBRARY:

  ```
  LIBRARY = CTC_array.o
  ```

- add the string "main" to EMBED_FUNCTION_NAME:

  ```
  EMBED_FUNCTION_NAME = main
  ```

  which will make that CTC++ will add (while producing the instrumented object files) the statement #pragma CTC APPEND immediately before each "exit point" of the main() function of the application.
  And when CTC++ meets this statement, it will call the ctc_append_all() function defined in CTC_array.c and so write the execution counters to a MON.txt file.

In fact, this last ctc_append_all() function calls another simpler ctc_send_char(char c) function, which you may have to write of your own in one of your source files (in prime example, it would rather be in prime.c file). This function takes one character for argument and writes it down in a MON.txt file, which would be done by following C/C++ code:

```
void ctc_send_char(char c)
{
  static FILE* fp = NULL;
  if (!fp) {
    fp = fopen("MON.txt","wb");
  }
  fputc(c,fp);
}
```

As you can see, for its first call, this function creates and opens a MON.txt file and then, since FILE* fp is declared *static*, continues to write characters in the same file.

However, this so produced MON.txt file is not the text file we would get by using the *CTC++ Host-Target add-on*: it just stores CTC_array[] final content, i.e. the execution counters' final version, without .

From now on, we just have left to convert this MON.txt text file to a MON.dmp binary dumped file by using the montxt2dmp CTC++ Bitcov utility as follows:

```
montxt2dmp -i MON.txt -o MON.dmp
```

For deeper information about `montxt2dmp` CTC++ Bitcov utility, refer to appendix A.4 at page 31.

To have a global point of view of this way of doing, you should refer to the following 2 flowchart.

3.3.3 Conclusion

Eventually, this is this `MON.dmp` binary file which now contains the execution counters as they were at the end of testes, and which will be the solely information transmitted from the Target computer to the Host computer. This process saves a lot of RAM on the Target: a `MON.dmp` represents as many octets as there are execution counters!

Next, with this `MON.dmp` binary file and the `MON.aux` file created at compilation time (which role is explained at next 3.4 section, "About `MON.aux` file and execution counters storage in `CTC_array[]`", page 18), we will be able to use `dmp2txt` CTC++ Bitcov utility (and then `ctc2dat` Testwell CTC++ utility) to produce the `MON.dat` file required to run CTC++ Postprocessor, as we will see in the next 3.5 section, "Recapitulation of CTC++ Bitcov execution process", at page 19.

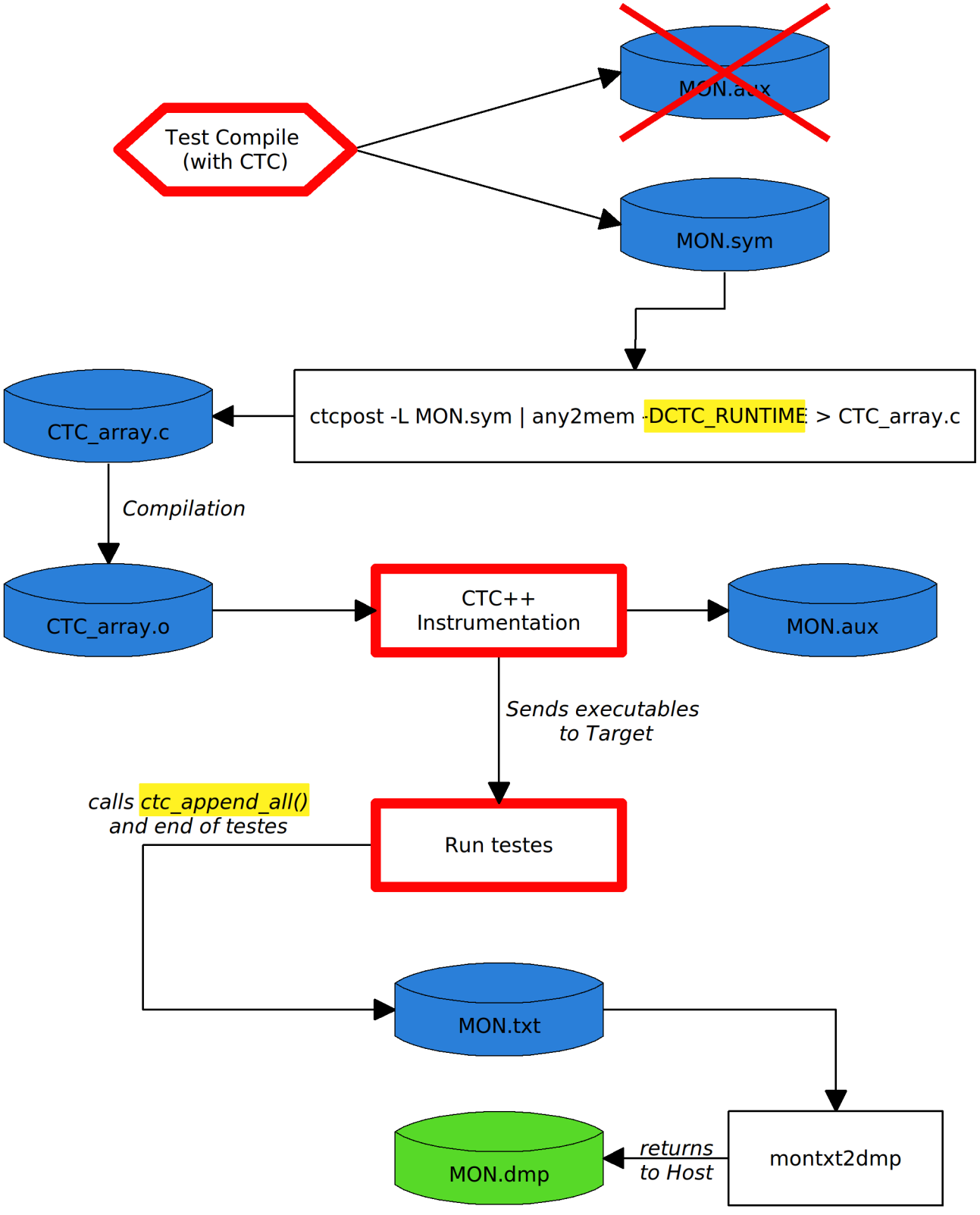

Figure 2: Execution counters saving using `any2mem` CTC++ Bitcov utility.

3.4 About `MON.aux` file and execution counters storage in `CTC_array[]`

You can see on the following 3 figure the correspondance existing beetween:

- one `MON.aux` file;
- each `CTC_MODULE_TAIL` environment variable present in each instrumented file.

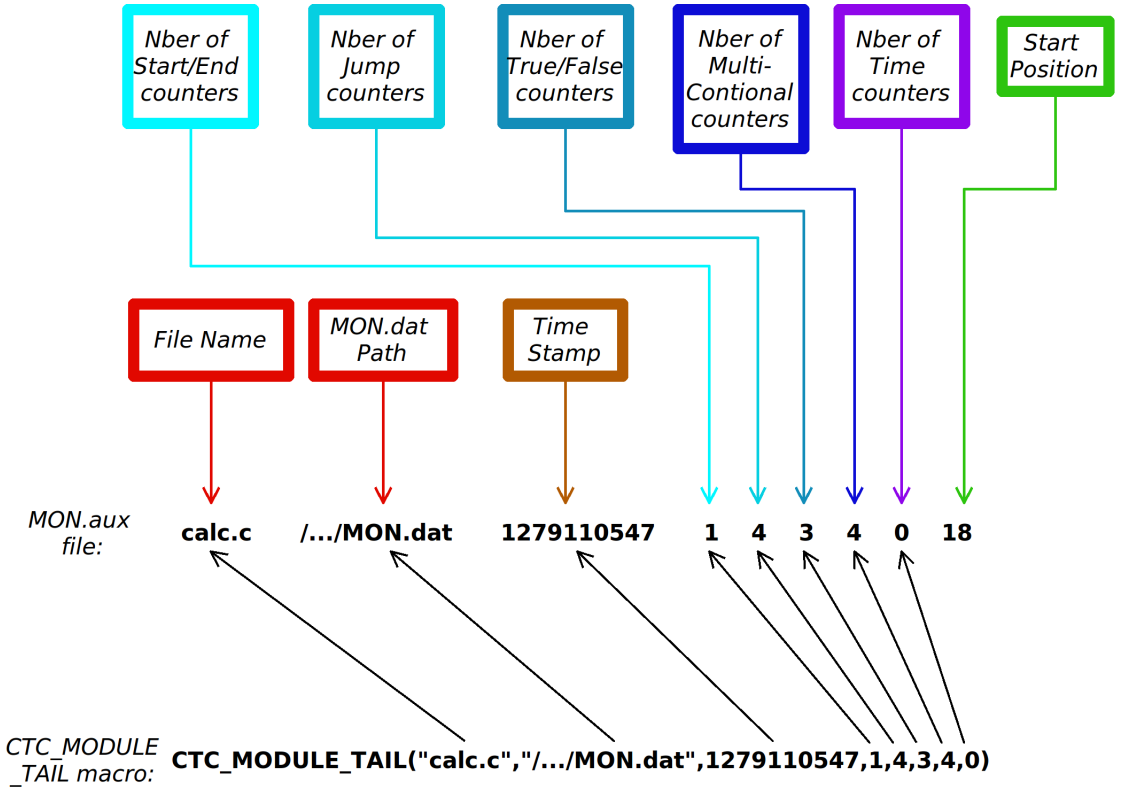

Figure 3: `MON.aux` content.

In fact, the `MON.aux` file stores all data which provide us to find easily in files storing the execution counters such and such counter type, counters concerning such and such source file and so on.

E.g. if we search which `CTC_array[]` part contains execution counters concerning the `calc.c` file of our `prime` example, we immediately know with `MON.aux` file that the first of these counters stays at 18^{th} box, and that the last stays at $18+2\times1+4+3\times2+4+0 = 34^{th}$ box, as illustrated by the following 4 picture:

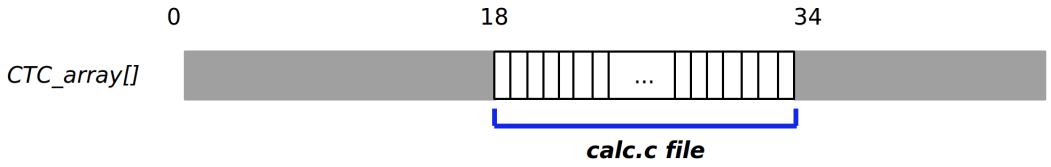

Figure 4: `CTC_array[]` content.

3.5 Recapitulation of CTC++ Bitcov execution process

To sum up the successive steps of CTC++ Bitcov execution process, we should refer to the following 5 diagram.

The first thing you have to do is to compute the instrumented executable files on the Host System. You should have a look to the User's Guide for more details about instrumentation with CTC++. In `prime` example's case, we dispose of a `makefile`, and so that we just have to type (with the coverage levels we want, teached by `-i` option):

```
ctcwrap -i m make
```

and then move the executable file(s) so produced to the Target System. If you want to learn more about the role of `MON.aux` produced during compilation/instrumentation, you should refer to appendix 3.4 (page 18, "About `MON.aux` file and execution counters storage in `CTC_array[]`").

Once you have:

- done the instrumentation (without forgetting the `-g` option of compilation if you want to use a Debug Interface),
- tested your application (through this Debug Interface or not),
- got your `MON.dmp` binary file, as explained in the previous 3.3 section (page 12, "Getting back the execution counters"),
- and sent it back to the Host,

there is just one thing left to do: convert it to a binary datafile, which we can call `MON.dat`, by using CTC++ Bitcov add-on functions:

```
dmp2txt -o MON.txt MON.dmp MON.aux
ctc2dat -i MON.txt
```

where the last command (`ctc2dat`) produces by default a `MON.dat` binary file the CTC++ Postprocessor can manipulate.

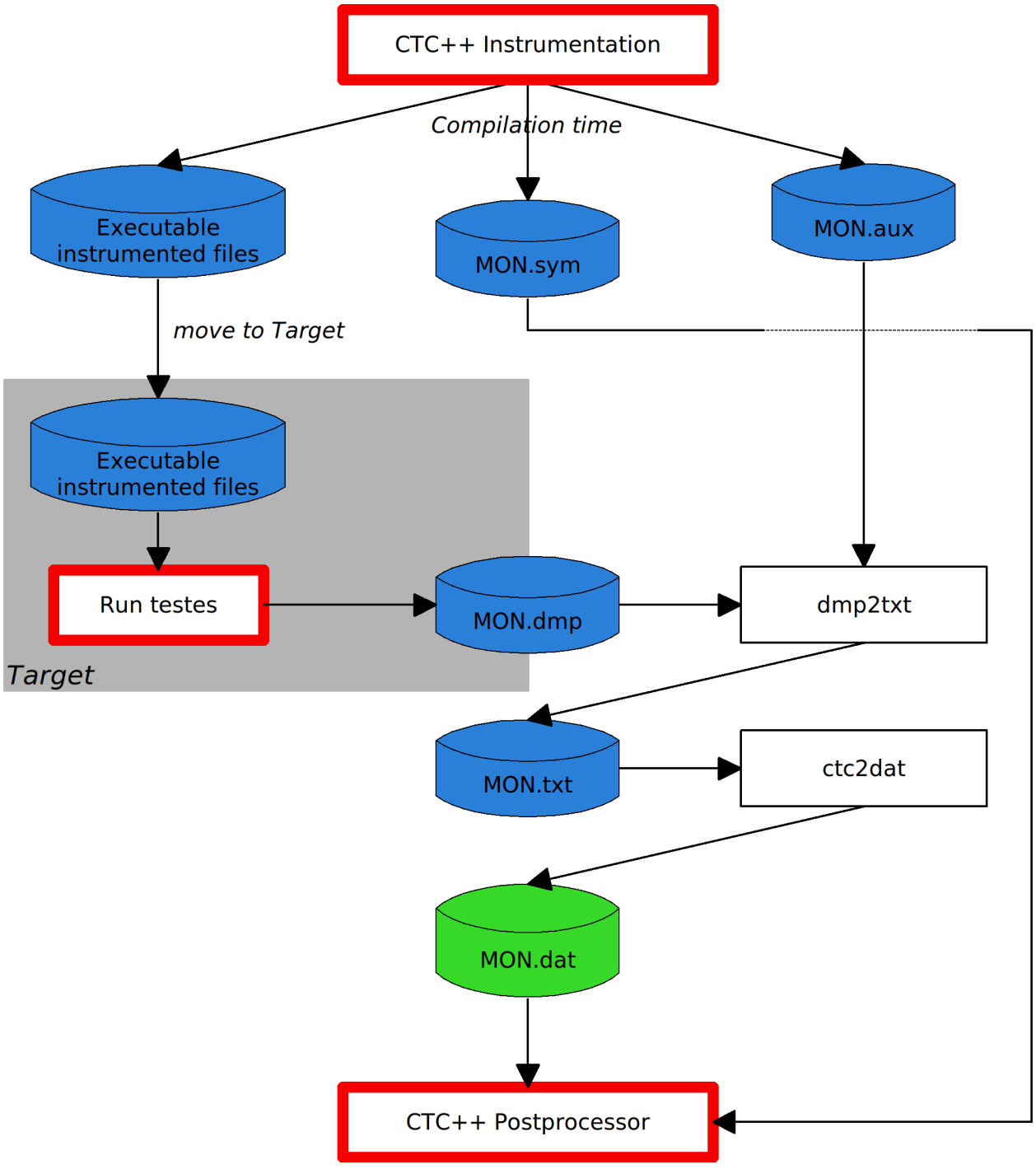

Figure 5: CTC++ Bitcov execution process.

At last, use CTC++ Postprocessor to produce, by the usual procedure, the Execution Profile Listing in the form of HTML pages:

```
ctcpost MON.sym MON.dat -p profile.txt
ctc2html -i profile.txt
```

These last commands will produce a HTML report in a created subdirectory `./CTCHTML`, which main page is `index.html`. There you will find the usual data, teaching you which parts of your program have been tested, where should be the "dead code" parts, and so on.

You can then find some sample outputs produced by CTC++ Postprocessor for `prime` example tested only with the 23 value. There is what you can find in `profile.txt` file produced by `ctcpost` utility:

```
*******************************************************************************
*           CTC++, Test Coverage Analyzer for C/C++, Version 6.5.6            *
*                                                                             *
*                          EXECUTION PROFILE LISTING                          *
*                                                                             *
*                     Copyright (c) 1993-2010 Testwell Oy                     *
*******************************************************************************


Symbol file(s) used   : MON.sym (Wed Jul 28 09:40:30 2010)
Data file(s) used     : MON.dat (Wed Jul 28 09:41:50 2010)
Listing produced at   : Wed Jul 28 09:42:20 2010
Coverage view         : As instrumented
```

Here you can observe that nothing has changed in the Execution Profile Listing header, which is still produced from `MON.sym` (containing the files' description) and `MON.dat` (containing the data about execution counters) files. You can also make the same statement about each file and function analysis, e.g.:

```
MONITORED SOURCE FILE : calc.c
INSTRUMENTATION MODE  : function-decision-multicondition

    START/       END/
      TRUE      FALSE    LINE DESCRIPTION
===============================================================================

         1          0       4 FUNCTION is_prime()
         0          1 -     8 if (val == 1 || val == 2 || val == 3)
         0            -     8   T || _ || _
         0            -     8   F || T || _
         0            -     8   F || F || T
                    1       8   F || F || F
         0            -     9   return 1
         0          1 -    10 if (val % 2 == 0)
```

```
         0             -    11   return 0
         1          1       12 for (;divisor < val / 2;)
         0          1 -     14   if (val % divisor == 0)
         0            -     15     return 0
         1                  17 return 1

***TER  47 % (  8/ 17) of FUNCTION is_prime()
-------------------------------------------------------------------------------


***TER  47 % (  8/ 17) of SOURCE FILE calc.c
-------------------------------------------------------------------------------
```

As we said previously, everything works as usual. Remember that we just tested `prime` with 23 value: we see here-before the path taken by this execution and, of course, multi-condition coverage is not satisfied only with this test...

Here is, for an informative purpose, `profile.txt` file's end:

```
MONITORED SOURCE FILE : io.c
INSTRUMENTATION MODE  : function-decision-multicondition

    START/       END/
      TRUE      FALSE    LINE DESCRIPTION
===============================================================================

         1           0       5 FUNCTION io_ask()
         0           1 -     11 if (( amount = scanf ( "%u" , & val ) ) <= 0)
         1                   14 return val

***TER  75 % (  3/  4) of FUNCTION io_ask()
-------------------------------------------------------------------------------

         1           1       18 FUNCTION io_report()

***TER 100 % (  1/  1) of FUNCTION io_report()
-------------------------------------------------------------------------------


***TER  80 % (  4/  5) of SOURCE FILE io.c
-------------------------------------------------------------------------------
```

```
MONITORED SOURCE FILE : prime.c
INSTRUMENTATION MODE  : function-decision-multicondition

    START/       END/
      TRUE      FALSE    LINE DESCRIPTION
=======================================================================

         1           0      11 FUNCTION main()
         1           1      15 while (( prime_candidate = io_ask ( ) ) > 0)
         1           0 -    17   if (is_prime ( prime_candidate ))
                            19   else
         0             -    23 return 0

***TER  67 % (  4/  6) of FUNCTION main()
-----------------------------------------------------------------------


***TER  67 % (  4/  6) of SOURCE FILE prime.c
-----------------------------------------------------------------------




SUMMARY
=======

Number of monitored source files  : 3
Number of source lines            : 63
Number of measurement points      : 30
TER                               : 57% (multicondition)
```

4 Walkthrough with `prime` example

For all following examples, assume we want to test `prime` example with the multiconditional instrumentation mode. Of course, typing the "`ls`" commands is not necessary, but they are there for a didactic purpose: mainly, it shows you at each step the files we dispose of, and files created by previous command.

4.1 Manual way (using a debugger)

Before typing following commands, you should have:

- added `ctc2static` to `RUN_AFTER_INSTR` in `ctc.ini` file (refer to 2.3 section, "Common adjustments", at page 9);
- declared an `unsigned char CTC_array[6] = 0;` at global scope of one of your source files (refer to 3.3.1 section, "Manual way", at page 12).

ls
calc.c calc.h io.c io.h makefile prime.c readme.txt

ctc -i m gcc -g *.c -o prime

ls
calc.c calc.h calc.o io.c io.h io.o makefile MON.aux MON.sym prime prime.c prime.o readme.txt

gdb ./prime
Reading symbols from ./prime...done.

(gdb) **b 19**
Breakpoint 1 at 0x804a49c: file prime.c, line 19.

(gdb) **r**
Starting program: ./prime
Enter a number (0 for stop program): **23**
23 IS a prime.

Enter a number (0 for stop program): **0**

Breakpoint 1, main() at prime.c:19
19 return 0;

(gdb) **dump binary value MON.dmp CTC_array**

(gdb) **c**
Continuing.

Program exited normally.

(gdb) **q**

ls
calc.c calc.h calc.o io.c io.h io.o makefile MON.aux MON.dmp MON.sym prime prime.c prime.o readme.txt

dmp2txt -o MON.txt MON.dmp MON.aux

ctc2dat -i MON.txt
Starting CTC++ Host-Target receiver.

Waiting for data
Receiving data
Data received
Appending modules
Done.

Waiting for data
End of "MON.txt" reached. Program ended.

ctcpost MON.sym MON.dat -p profile.txt

ctc2html -i profile.txt
Sorting execution profile listing
Directory 1 Done
Generating HTML report
File 1 2 3 Done

4.2 Automatic transfer (using `any2mem` utility)

Before typing following commands, you should have:

- added `ctc2static` to `RUN_AFTER_INSTR` in `ctc.ini` file (refer to 2.3 section, "Common adjustments", at page 9);
- added `main` to `EMBED_FUNCTION_NAME` in `ctc.ini` file (refer to 3.3.2 section, "Using `any2mem` CTC++ Bitcov utility", at page 13).

ls
calc.c calc.h io.c io.h makefile prime.c readme.txt

ctc -i m gcc -g *.c -o prime

ls
calc.c calc.h calc.o io.c io.h io.o makefile MON.aux MON.sym prime.c prime.o readme.txt

ctcpost -L MON.sym | any2mem -DCTC_RUNTIME > CTC_array.c

ls
calc.c calc.h calc.o CTC_array.c io.c io.h io.o makefile MON.aux MON.sym prime.c prime.o readme.txt

gcc -c CTC_array.c

ls
calc.c calc.h calc.o CTC_array.c CTC_array.o io.c io.h io.o makefile MON.aux MON.sym prime.c prime.o readme.txt Target

Then, add the so produced `CTC_array.o` file to `LIBRARY` in `ctc.ini` file (refer to 3.3.2 section, "Using `any2mem` CTC++ Bitcov utility", at page 13).

rm MON.aux

ctc -i m gcc -g calc.c io.c prime.c -o prime

ls
calc.c calc.h calc.o CTC_array.c CTC_array.o io.c io.h io.o makefile MON.aux MON.sym prime prime.c prime.o readme.txt

./prime
Enter a number (0 for stop program): **23**
23 IS a prime.

Enter a number (0 for stop program): **0**

ls
calc.c calc.h calc.o CTC_array.c CTC_array.o io.c io.h io.o makefile MON.aux MON.txt MON.sym prime prime.c prime.o readme.txt

montxt2dmp -i MON.txt -o MON.dmp
Starting CTC++ host-target receiver.

Waiting for data
OK, dumping data

Waiting for data

dmp2txt -o MON.txt MON.dmp MON.aux

ctc2dat -i MON.txt

Starting CTC++ Host-Target receiver.

Waiting for data
Receiving data
Data received
Appending modules
Done.

Waiting for data
End of "MON.txt" reached. Program ended.

ctcpost MON.sym MON.dat -p profile.txt

ctc2html -i profile.txt

Sorting execution profile listing
Directory 1 Done
Generating HTML report
File 1 2 3 Done

5 How to work with real (32-bits) counters

Default counters are 8-bits integer counters, but if you want to run CTC++ Bitcov add-on on bigger targets, this size will probably not be sufficient.

Running CTC++ Bitcov add-on on bigger targets using big/real counters with a size of 32 bits is possible, but requires some additional statements during its execution process. Mainly, these are the following:

1. Before having done anything, you have to modify your `ctc.ini` file with adding the `-DCTC_NO_BITS` compilation option as follows:

   ```
   OPT_ADD_COMPILE = -DCTC_NO_BITS
   ```

   This option defines the `CTC_NO_BITS` macro, and so enables you to use real counters to compile.

2. This `-DCTC_NO_BITS` option should also be passed to `any2mem` utility when producing the `CTC_array.c` file:

   ```
   ctcpost -L MON.sym | any2mem -DCTC_NO_BITS > CTC_array.c
   ```

   It will define the `CTC_NO_BITS` environment variable in `CTC_array.c` script, and so alows you to use the functions and definitions provided by it (especially, it will define the `CTC_array[]` with the correct dimension).

3. Eventually, when using `dmp2txt` CTC++ Bitcov utility, you have to add an option specifying the execution counters' format you are using. This should be done by:

   ```
   dmp2txt -32 -o MON.txt MON.dmp MON.aux
   ```

   For more details about this kind of `dmp2txt` options, refer to appendix A.3, "Man page: `dmp2txt` utility", page 29.

For further information about using real counters with CTC++ Bitcov add-on, you should also refer to `readme-big.txt` and `readme-big.html` files, which can be found in documentation folder of CTC++ Bitcov add-on installation directory.

A CTC++ Bitcov utilities man pages

A.1 Man page: any2mem utility

```
NAME
       any2mem - Calulates needed CTC_array, especially size

SYNTAX
       any2mem

DESCRIPTION
       This program reads  ctcpost -L  output and  caluclates  the  needed
       CTC_array. The output is printed as compilable code snippet.

EXAMPLES
       To run this program the standard way type:

       ctcpost -L MON.sym | any2mem > ctc_array.c
```

A.2 Man page: ctc2static utility

```
NAME
       ctc2static - RUN_AFTER_INSTR script

SYNTAX
       ctc2static

DESCRIPTION
       This script has to  only  called  by CTC++ as  a  RUN_AFTER_INSTR
       script. Not to invoke directly!

       You can set in this script $nogui=1 to one  to  prevent  poping  up
       error messages.
```

A.3 Man page: dmp2txt utility

```
NAME
       dmp2txt - Converts memory dump to text, as expected by ctc2dat

SYNTAX
       dmp2txt [options] MON.dmp MON.aux
       dmp2txt [options] MON.hex MON.auc
```

```
DESCRIPTION
       This program converts a binary, or Intel hex or  Motorola s-record,
       or Tektronix, dumped CTC_array based on the  corresponding  MON.aux
       file to the serial stream normaly transmitted from the target.

FILES
       MON.dmp The binary dumped CTC_array

       MON.hex The  Intel hex,  Motorola  s-record,  or  Tektronix  dumped
       CTC_array

       MON.aux The bookkeeping filing created during CTC++ compilation by
       the ctc2static script

EXAMPLES
       To run this program the standard way type:

        dmp2txt MON.dmp MON.aux > MON.txt
       or
        dmp2txt MON.hex MON.aux > MON.txt

       Alternativly you can pipe output to the ctc2dat application:

        dmp2txt MON.dmp MON.aux | ctc2dat
       or
        dmp2txt MON.hex MON.aux | ctc2dat

OPTIONS
       --help -h
       Displays brief usage informations

       --version
       Displays version information

       --dump2
       For debug purposes only.  Dumps the  input data to file  in  binary
       format.

       --pad hexnumber
       Rarely used. Setts padding info for dumped files.

       The following only for non-bit (real counters), when using:
       CTC_NO_BITS in OPT_ADD_COMPILE=

       -l8, -l16, -l32, -l64, -l128
       Dump was created on a little endian system,  using  the  given  bit
```

amount for CTC_COUNTER. E.g. use -l32 on a x86 system using Linux or Windows.

-b8, -b16, -b32, -b64, -b128
Dump was created on a big endian system, using the given bit amount for CTC_COUNTER. E.g. use -b32 on a x86 system using Linux or Windows.

A.4 Man page: montxt2dmp utility

NAME

montxt2dmp - Converts MON.txt file to memory dump file, as expected by dmp2txt.

SYNTAX

montxt2dmp [-i inputfile] [-o outputfile] [other options]

DESCRIPTION

This program converts a textual MON.txt file, which holds CTC_array final content, to a binary MON.dmp file dumping CTC_array one to one as it was at test end.

FILES

MON.txt - The textual dumped CTC_array, created just after testes by script contained in CTC_array.c file.
Its content should look like:
<start:"CTC_array size (1 or 2 integers)","CTC_array content">.

MON.dmp - The binary dumped CTC_array.

OPTIONS

-i inputfile.txt
Input file name. Default is piped stdin.

-o outputfile.dat
Output file name. Default is MON.dmp file produced in current directory.

-b Brief. Detailed information of the transfer is not printed.

-s Silent. Do not print any information.

-pad hexnumber
Rarely used. Setts padding info for dumped files.

B Table of -DCTC_* macro definitions

Macro definition	Description
-DCTC_H	prevents any2mem from putting in CTC_array.c file header declarations of functions it contains. If CTC_NO_BITCOV_CTC_INIT is already defined, CTC_H takes not any effect.
-DCTC_NO_BITCOV_CTC_INIT	prevents any2mem from defining a new ctc_init() function in CTC_array.c file. This function is already defined by default by Testwell CTC++ and permits you to set all values of CTC_array[] to zero. This macro also forces the definition of CTC_NO_RUNTIME and prevents the use of CTC_H.
-DCTC_NO_BITS	enables the use of real (32-bits) counters, by defining automatically the CTC_array[] with the good size and the good data format. Refer to previous 5 section ("How to work with real (32-bits) counters", page 28) to find further information.
-DCTC_RUNTIME	enables the use of functions generated by any2mem in CTC_array.c file, especially ctc_append_all(). If CTC_NO_BITCOV_CTC_INIT is already defined, CTC_RUNTIME also.
-DCTC_SAFE	prevents execution counters from overflowing (for 8-bits counters).

List of Figures

1 Manual execution counters saving. 14
2 Execution counters saving using `any2mem` CTC++Bitcov utility. 17
3 `MON.aux` content. 18
4 `CTC_array[]` content. 19
5 CTC++Bitcov execution process. 20

Glossary

Cross-compiler Compiler capable of creating executable code on a *Host* system for a *Target* system. See at http://en.wikipedia.org/wiki/Cross_compiler, 11

ctc.ini file Default CTC++ configuration file, containing all options used by CTC++ and all of its utilities during their execution, 9

CTC_array[] Array storing current execution counters during testes, 5

Debug Interface Software providing test and debug of other programs, by showing current variables' values during execution, setting breakpoints in programs, and so on. See at http://en.wikipedia.org/wiki/Debugger, 12

Execution counters Numbers measuring how many times each measurement point of the instrumented code has been executed untill now. They are stored in the *CTC_array[]*, 4

Execution Profile Listing Report resulting of test session(s), showing for each source file and each function how high is the TER, what lines of source code have been exercised / not exercised, etc., 11

Host-Target usage of CTC++ Use of an intermediary sytem (*Host*) to instrument/compile/link the source code, then send it to the *Target* system to run testes, and finally bring final execution counters back to the *Host* system, producing there the Execution Profile Listing. Have a look to the **Testwell CTC++ User's Guide** at section "Host-Target Testing", 4

"Huge Targets" approach Refer to Real (32-bits) counters, 13

Instrumentation Process producing, from source files, instrumented files in which some additional statements (*probes*) are added in some places that are relevant from test coverage measuring point of view, 4

MON files — MON is an acronym for *Monitor*: such files contain accurate data (computed by CTC++) about the application you are testing, its instrumentation, testes run until now, etc., 8

Real (32-bits) counters — Bigger execution counters format (32 bits instead of 8), allowing users to make more testes before counters overflow, 13

TER — Test Effectiveness Ratio, a test coverage measure (%) calculated for each instrumented function, source file and the whole program and showing how many percent of instrumented source code have already been tested, 4

UAC — User Account Control. See at http://en.wikipedia.org/wiki/User_Account_Control, 5

Wine — Software providing use of Windows programs on Unix/Linux. See at http://en.wikipedia.org/wiki/Wine_(software), 5

Index

B

Bitcov utilities
- any2mem 5, 8, 13, 28, 29
- ctc2static 5, 8, 9, 12, 24, 25, 29
- dmp2mon 6
- dmp2txt 6, 8, 9, 28, 29
- hex2bin 6
- hostmain 5
- montxt2dmp 7, 8, 15, 31
- mot2bin 6

C

Coverage levels
- Condition coverage 5
- Decision coverage 4
- Function coverage 4
- Modified condition/decision coverage . 5
- Multicondition coverage 5, 22, 24

Cross-compiler 11, 15
CTC++ Postprocessor 11–13, 16, 21
ctc.ini file 10, 12, 15, 24, 25, 28
CTC_array.c file 13, 15, 28
CTC_array[] 7, 12, 16, 24, 28

D

Debug Interface 12, 19

E

Execution counters 4, 11–13, 15, 16, 21
Execution Profile Listing 11, 21

F

Functions in CTC_array.c file
- ctc_append_all() 15
- ctc_init() 7
- ctc_send_char(char c) 15

H

Host-Target usage of CTC++ 4, 11, 15
"Huge Targets" 8, 13, 28

M

Macros
- CTC_H 32
- CTC_MODULE_TAIL 16
- CTC_NO_BITCOV_CTC_INIT 32
- CTC_NO_BITS 13, 28, 32
- CTC_RUNTIME 13, 32
- CTC_SAFE 5, 32

Microcontrollers 4, 11
MON.aux file 6, 9, 16
MON.dat file 20, 21
MON.dmp file 13, 16, 20
MON.sym file 12, 13, 21

R

Real (32-bits) counters 13, 28

S

Scripts in ctc.ini file
- EMBED_FUNCTION_NAME . . . 15, 25
- LIBRARY 15
- OPT_ADD_COMPILE 28
- RUN_AFTER_INSTR 8, 10, 24, 25

T

TER 4

U

UAC 6

W

Wine 7

Softwaretest- und Analysetools

Technologiepark, In der Spöck 10
D-77656 Offenburg
Germany